Report - Praxisblatt 1

Malte A. Weyrich
Jan P. Hummel

7. Mai 2024

1 Aufgabe

Für einen Vector C der Länge n macht der naive Ansatz ganze:

$$\frac{n^3 + 3n^2 + 2n}{6}$$

Additionen. Für unser n wären das $\approx 4.11 \times 10^{17}$ Additionen. Nehmen wir an, der Rechner würde eine Addition innerhalb von $0.01ms$ abschließen en, dann wäre:
$4.11 \times 10^{17} \cdot 0.01ms \approx 4 \times 10^{15}ms = 4 \times 10^{12}s \approx 130045$ Jahre. Also wäre das Problem unlösbar für die gegebene Hardware.

Anmerkung: Additionen dauern je nach größe der zu addierenden zahlen oft unterschiedlich lang, das obige ist eine grobe Schätzung.

Der rekursive Ansatz wird genau wie der naive Ansatz zu lange dauern, der Unterschied der Komplexitäten ist zwar da, jedoch skalieren beide Algorithmen auf diesem großen Input schlecht. Der dominante Term bleibt weiterhin n^3.

Zudem kommt es bei dem rekursiven Ansatz auch zu Problemen mit dem Stack: Der Algorithmus hat enorm vile Methodenaufrufe und es kann sehr gut sein, dass es hierbei zu einem StackOverflow kommen könnte.

Der Ansatz hat eine *Komplexität* von

$$\frac{n^2 + n}{2}.$$

In diesem Ansatz bedienen wir uns eines Arrays, welches bereits berechnete Ergebnisse zwischenspeichert und somit Redundante Berechnungen verhindert. Es ist davon auszugehen, dass die Werte in *ints* gespeichert werden (1 *int* = 4 Bytes = 32 Bit). Dieses 2d Array ist insgesamt $n \times n$ groß, wenn man davon ausgeht, dass das Array am Anfang des Ausführend des Programms basierend auf dem Eingabe Wert Initialisiert wird.

Der Algorithmus benutzt zwar nur die Hälfte der Matrix, jedoch ist die andere (ungenutzte) Hälfte der Matrix ebenfalls mit *ints* initialisiert worden (es sei denn der Algorithmus Initialisiert die Matrix optimal, was nicht aus dem *pseudo-code* hervorgeht). Für das Beispiel gehen wir davon aus, dass die Matrix vollständig, mit der Größe $n \times n$ initialisiert wurde:

Hierbei sind $\times$ genutzte *ints* und alles unter der Diagonalen sind *ints* die mit 0 initialisiert wurden. Die größe beträgt $2.47 \times 10^6 \times 2.47 \times 10^6$. In jeder der Zellen steht eine Adresse auf einen reservierten, 32 Bit großen Bereich. Also insgesamt $\left(2.47 \times 10^6\right)^2 \cdot 32Bit = 6.1009 \times 10^{12} \cdot 32Bit = 1.952288 \times 10^{14} \approx 22727.62GB$ welches $> 1.2GB$ ist. Also passt das Array nicht in den Arbeitsspeicher.

Divide and Conquer hat eine Komplexität von:

$$n \cdot \log n - 2n + 2$$

Was für unsere Eingabe n und unserem geschätzten Wert pro Addition $(0.01ms)$ also:

$$\approx 10849963 \cdot 0.01ms \approx 108499ms \approx 1.8min.$$

Zeitlich ist es also kein Problem.
Lediglich bei der rekursiven Zerlegung könnte es vielleicht auch zu Problemen bezüglich des Stacks kommen.

Optimaler Ansatz:
Da der optimal Ansatz das Problem in linearer Zeit ($\mathcal{O}(n)$) löst, gibt es keine Komplikationen bezüglich des Ausführens des Algorithmus.

Bei einer Eingabe C^n und mit unserer Schätzung von $0.01ms$, beträgt die Laufzeit:

$$2.47 \times 10^6 \cdot 0.01ms \approx 24s$$

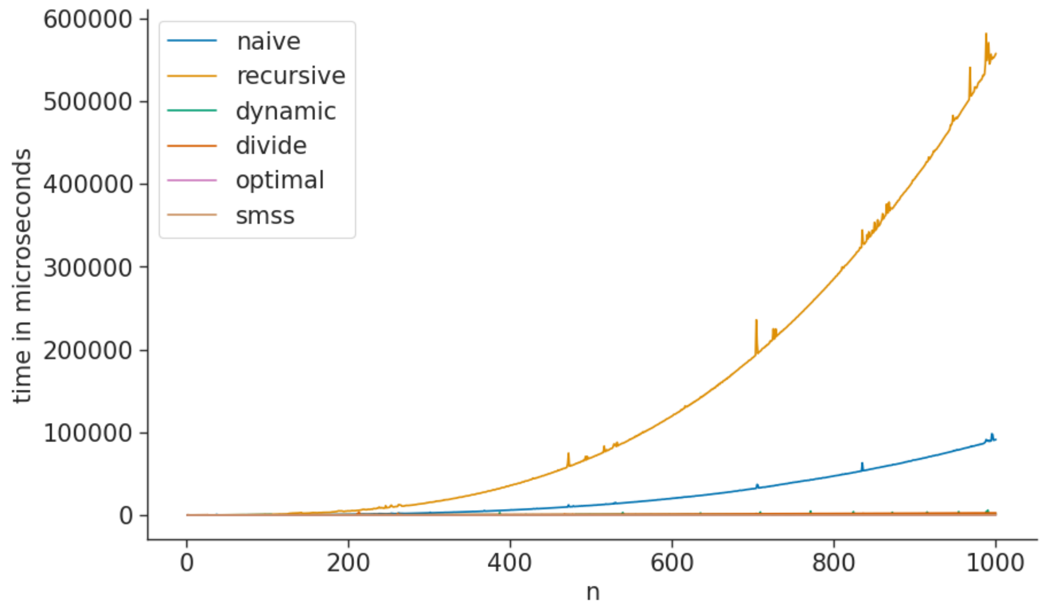

Abbildung 1: Überblick aller Algorithmen für Eingabeintervall [1; 1000]

2 Aufgabe

2.1 2a - SMSS

Wir erweitern den optimalen Ansatz, indem wir jedes gefundene *segment* in einer *ArrayList< ArrayList<int[3]>* > abspeichern. Die äußere Liste initialisiert für jeden neuen *max_score* der gefunden wurde eine neue innere Liste. Diese innere Liste wird so lange mit gleichwertigen *segmenten* befüllt, bis ein *segment* gefunden wurde, mit einen größerem *max_score*. So ist die Liste von links nach rechts aufsteigend nach *max_score* sortiert (d.h. die letzte innere Liste beinhaltet alle *MSS*). Am Ende wird einmal über die am *Endindex* stehende Liste in $\mathcal{O}(n)$ iteriert, um die tatsächlichen non overlapping *MSS* in einer finale Liste zu speichern.

Folgende Fälle werden abgedeckt:

```
CASE 1:
1: ---+===+-----------
2: ---+=====+---------
```

Hier darf das $segment_2$ nicht ausgegeben werden, da $segment_1 \subseteq segment_2$. Also merken wir uns jeweils den Start- und Endindex des letzten *segments*. So können wir die letzten Indizes mit den momentanen vergleichen und **CASE 1** abfangen.

```
CASE 2:
1: ---+=====+---------
2: ---+===+-----------
```

CASE 2 ist nicht möglich, da $segment_2$ vor $segment_1$ in der Liste abgespeichert ist. Der Algorithmus ist so konzipiert, dass er wie bei **CASE 1** erst das kürzere $segment_1$ findet und dann das längere $segment_2$. Also müssen wir uns nicht um **CASE 2** kümmern.

```
CASE 3:
1: -----+=+-----------
2: ---+=====+---------

CASE 4
1: ---+=====+---------
2: -----+=+-----------
```

CASE 3 & 4 Sind ebenfalls nicht möglich, da *MSS* entweder den gleichen Start- oder Endindex haben.

```
CASE 5:
1: -----+===+---------
2: ---+=====+---------
```

Hier wird sich in der *i-ten* Iteration die Start- und Endindizes von $segment_1$ gemerkt und demnach das $segment_2$ in der *i+1-ten* Iteration nicht dem Endresultat hinzugefügt, da der Endindex von $segment_2 \ngtr$ Endindex von $segment_1$.

2.2 2b - SMSS mit minimaler Länge

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

2.3 2c - Alle MSS

Für die 2c haben wir unseren ursprünglichen Ansatz nicht weiter verwendet. Der *optimale Algorithmus* ist in sich selbst so effizient, dass er null Folgen am Anfang und am Ende von Segmenten direkt ignoriert. Wir haben uns also überlegt, welcher der anderen Algorithmen mit eine möglichst guten Laufzeit, diese Fälle auch betrachtet. Wir haben uns dazu entschieden den *dynamic programming Algorithmus* zu erweitern. Dieser hat nämlich den Vorteil bereits ohne Erweiterungen, alle Kombinationen von Teilintervallen zu betrachten. Insbesondere die Teilintervalle, die der *optimale Algorithmus* nicht in Betracht zieht.

Beispiel

```
                        5 -5 9 -3 13 -5 5 -5 5
optimal             :   -----+=====+----------

tatsächliche MSS    :   -----+=====+----------
                        -----+=========+-----
                        -----+==============+
                        +=========+----------
                        +==============+-----
                        +===================+
```

Der *dynamic programming Algorithmus* betrachtet in dem oberen Beispiel alle der Teilfolgen. Also haben wir wieder wie in TODO eine äußere Liste mit inneren Listen erstellt, und für jeden neuen *maxscore* eine neue innere Liste erstellt, die dann mit allen gleichwertigen Segmenten (also allen Segmenten mit demselben *maxscore*) befüllt wird, bis ein größerer *maxscore* gefunden wird. So findet der *2_c Algorithmus* also definitiv alle MSS, denn wir wissen bereits, dass der *dynamic programming* Ansatz korrekt ist und alle Kombinationen in Betracht zieht. Jedoch bußt dieser Ansatz bei der Speicherkomplexität ein, das *int[][]* Array wächst zu einem gegebenen Eingabe Vektor mit Länge n exponentiell: n^2. D.h. für große Eingaben verbrauchen wir enorm viel Speicherplatz.

Wie bereits in *1. Theorie Blatt Aufgabe 2* gezeigt, benutzt der *dynamic programming Algorithmus* immer nur die Hälfte des Arrays. Allgemein hat das Array die folgende Form:

$$\begin{pmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & 0 & & \times & \times \\ & & & & \times \end{pmatrix}$$

Beispiel

Betrachten wir die jeweiligen Endzustände der Datenstrukturen von Algorithmus *2_c* und der Arbeitsspeicher schonenden Variante *2_c_1*:

- Eingabe: $v = \{5, -5, 9, -3, 13, -5, 5, -5, 5\}$
- Hardware: (*Processor: AMD® Ryzen 7 pro 4750u with radeon graphics × 16; Memory: 16.0 GB*)

```
Array S[n][n] of Default Dynamic Programming (2_c) on v:
   0: [5 ,0 ,9 ,6 ,19,14,19,14,19]
   1: [0 ,-5,4 ,1 ,14,9 ,14,9 ,14]
   2: [0 ,0 ,9 ,6 ,19,14,19,14,19]
   3: [0 ,0 ,0 ,-3,10,5 ,10,5 ,10]
   4: [0 ,0 ,0 ,0 ,13,8 ,13,8 ,13]
   5: [0 ,0 ,0 ,0 ,0 ,-5,0 ,-5, 0]
   6: [0 ,0 ,0 ,0 ,0 ,0 ,5 , 0, 5]
   7: [0 ,0 ,0 ,0 ,0 ,0 ,0 ,-5,0 ]
   8: [0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,5 ]

Max n on given Hardware: 30808:
java -jar ... --algorithms 2_c --size 30808

ArrayList S containing int[] Arrays of Improved Dynamic Programming (2_c_1) on v:
   0: [5 ,0 ,9 ,6 ,19,14,19,14,19]
   1: [-5,4 ,1 ,14,9 ,14,9 ,14]
   2: [9 ,6 ,19,14,19,14,19]
   3: [-3,10,5 ,10,5 ,10]
   4: [13,8 ,13,8 ,13]
   5: [-5,0 ,-5,0]
   6: [5 ,0 ,5]
   7: [-5,0]
   8: [5]

Max n on given Hardware: 44069:
java -jar ... --algorithms 2_c_1 --size 44069
```

Der *improved dynamic programming 2_c_1* hat keine Position in der Datenstruktur, die ungenutzt bleibt. Der verbesserte Ansatz schafft es 13261 zusätzliche Zahlen zu verarbeiten, also ein $\approx$ 43% größeres n als der *default 2_c Algorithmus* (*auf der gegebenen Hardware). An der Komplexität hat sich durch die Abänderungen nichts geändert. In Abbildung 2 werden die zwei Ansätze in ihrer Laufzeit verglichen. Die Spikes in der Abbildung entstehen durch die inkonsistente Geschwindigkeit bei der Array Initialisierung.

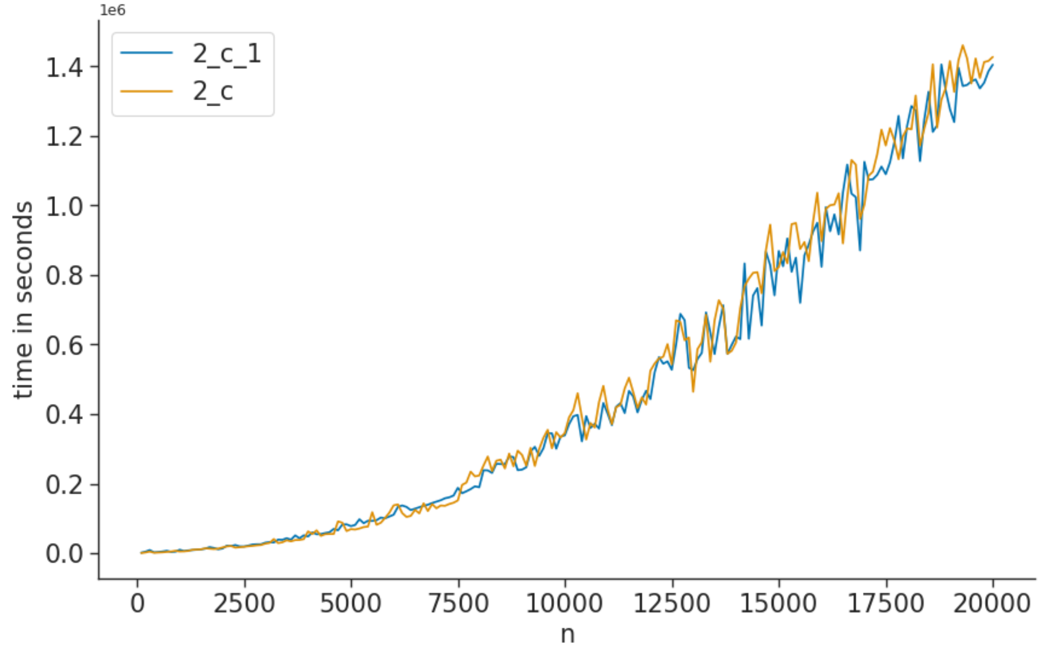

Abbildung 2: Ansatz 2_c verglichen mit dem Arbeitsspeicher optimierten Ansatz 2_c_1

2.4 Ergebnisse

Um die Ergebnisse auf dem Beispiel des Übungsblattes zu rekreiren ruft man die *JAR* folgendermaßen auf:

```
java -jar JAR --v 5 -2 5 -2 1 -9 12 -2 24 -5 13 -12 3 -13 5 -2 -1 2

// Naive:
//
//  [6,10] mit score 42
//  466 µs
//  for input size 19

// Recursive:
//
//  [6,10] mit score 42
//  180 µs
//  for input size 19

// Dynamic Programming:
//
//  [6,10] mit score 42
//  29 µs
//  for input size 19

// Divide and Conquer:
//
//  [8,8] mit score 24
//  847 µs
//  for input size 19

// Optimal:
//  [6,10] mit score 42
```

```
//  4 µs
//  for input size 19

// 2_a (MSS):
//
//  [6,10] mit score 42
//  24 µs
//  for input size 19

// 2_b (SMSS):
//
//  [6,10] mit score 42
//  24 µs
//  for input size 19

// 2_c (All SMSS):
//
//  [6,10] mit score 42
//  32 µs
//  for input size 19

// 2_c_1 (All SMSS & Optimized space usage):
//
//  [6,10] mit score 42
//  49 µs
//  for input size 19
```